kodefox

27th March 2017

Modern Concurrency: async and await

@sstur_

kodefox

I'm Simon

JavaScript and React Enthusiast

Founder at KodeFox

Twitter: @sstur_

kodefox

Today we'll talk about Concurrency in Modern JavaScript.

kodefox

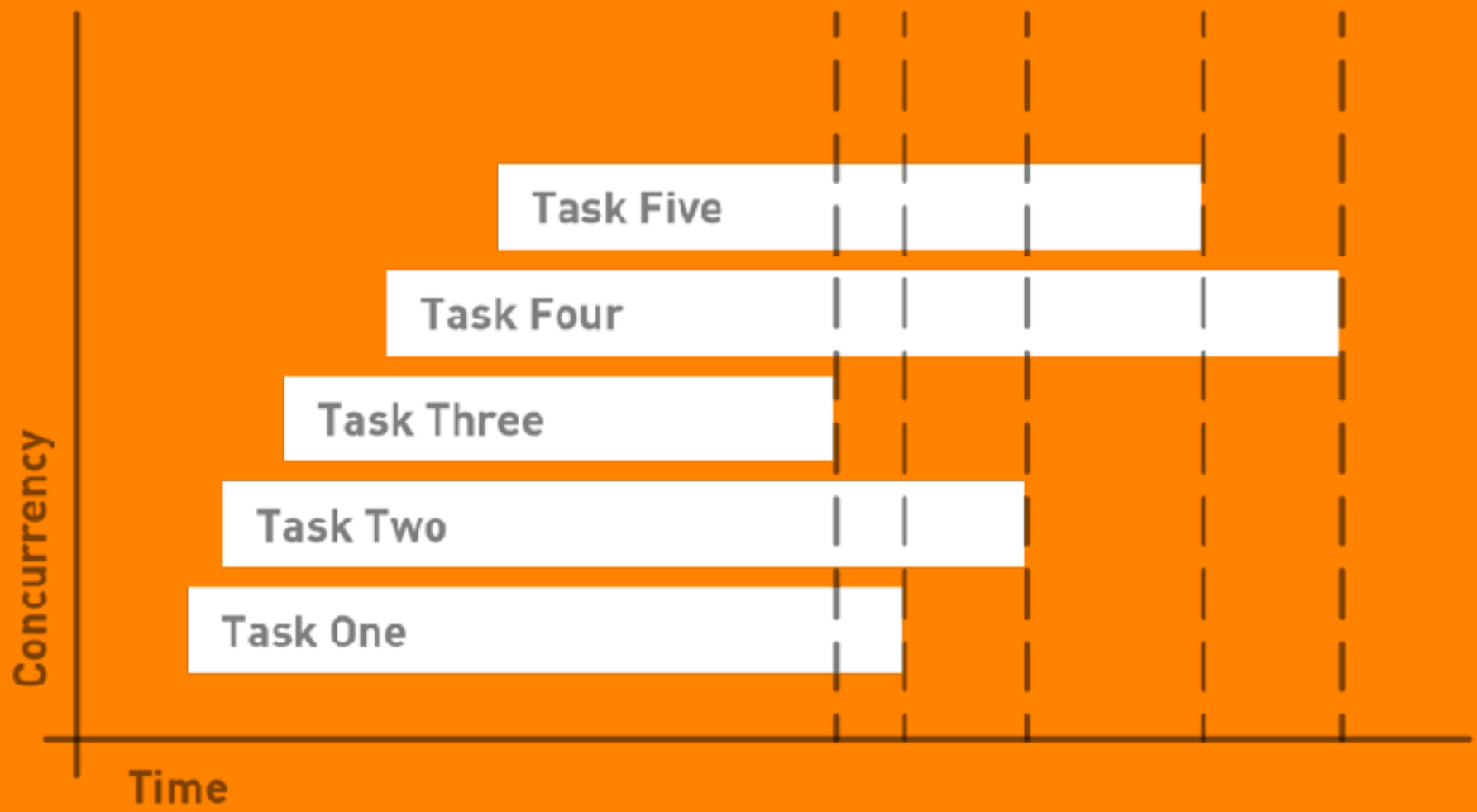
Concurrency

Doing multiple tasks in a period of time.

(Generally order-independent or partially-ordered units of work)

kodefox

Concurrency



Concurrency is important when waiting on input/output such as network requests, reading/writing from disk, or user input.

kodefox

Typically, two ways programs wait for IO:

- blocking (“synchronous”)
  - Easy to write
  - Uses multi-threading
  - Memory and context-switching overhead
- non-blocking/event-loop (“asynchronous”)
  - Single-threaded
  - High-concurrency with low-memory consumption
  - Great for UI and IO-bound services rather than CPU-bound

kodefox

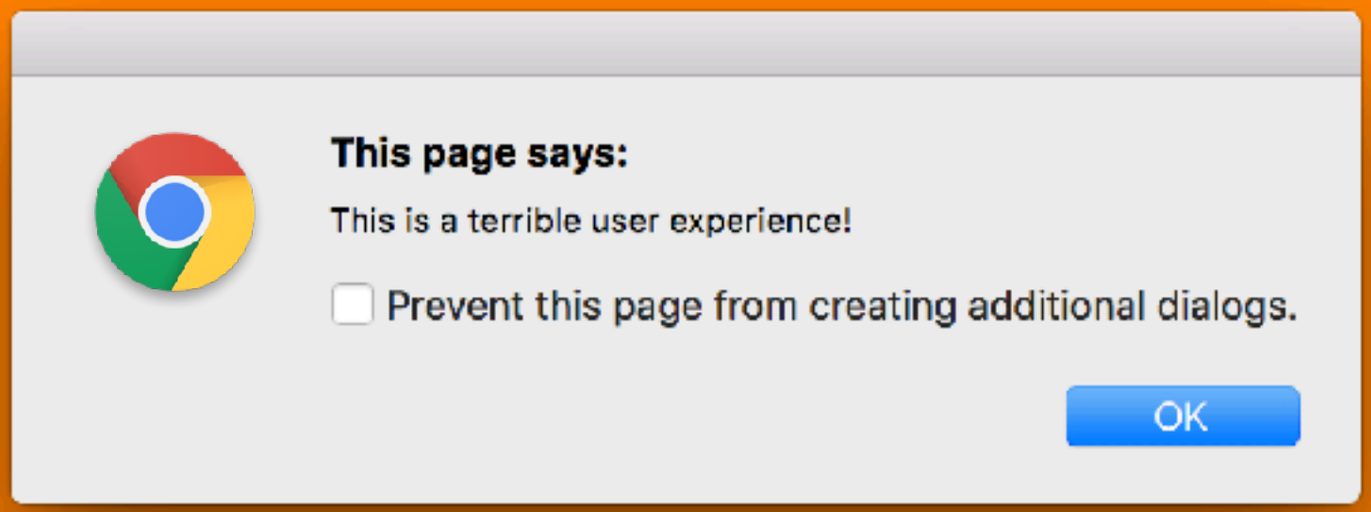
All modern JavaScript engines use the non-blocking/event-loop approach.

kodefox

So what happens if you block in JavaScript?

There are a few things that will block:

- alert/prompt/confirm
- synchronous XMLHttpRequest (rare)
- fs.readFileSync and friends in Node



Blocking in the browser will halt everything, including all user interaction, even scrolling.

kodefox

Callbacks

Just pass a function that will be called when the task is complete.

kodefox

Callbacks

```
// Callback style
readFile('file.txt', (content) => {
  console.log(content);
});
```

kodefox

Callbacks

Pros:

- **Great low-level abstraction**
- **Performant with low overhead**
- **Can do almost any async task with callbacks**

kodefox

Callbacks

Cons:

- **Doing things in sequence is hard. Doing things in parallel is harder!**
- **Give up constructs such as for/while and try/catch**
- **Error handling is difficult**
- **Code readability suffers and systems become hard to maintain**

kodefox

```
// Every day on StackOverflow
function getUserName() {
  let name;
  $.get('/users/123', (user) => {
    name = user.name;
  });
  return name;
}

// Why doesn't it work??
console.log('User Name:', getUserName());
```

kodefox

Callbacks Add Complexity

It's messy to chain tasks and difficult to do parallelism.

```
// Sequential tasks (even without error handling)
function getTotalFileSize(file1, file2, file3, callback) {
  let total = 0;
  stat(file1, (error, info) => {
    total += info.size;
    stat(file2, (error, info) => {
      total += info.size;
      stat(file3, (error, info) => {
        total += info.size;
        callback(total);
      });
    });
  });
}
```

```
// Parallel tasks (without error handling)
function getTotalFileSize(file1, file2, file3, callback) {
  let numFinished = 0;
  let total = 0;
  [file1, file2, file3].forEach((file) => {
    stat(file, (error, info) => {
      total += info.size;
      numFinished += 1;
      if (numFinished === 3) {
        callback(total);
      }
    });
  });
}
```

kodefox

Plus Error Handling

We spend a lot of effort checking if an async task failed.

We completely lose try/catch.

```
// This is not even good Error handling.
stat(file1, (error, info) => {
  if (error) {
    console.error(error);
    return;
  }
  total += info.size;
  stat(file2, (error, info) => {
    if (error) {
      console.error(error);
      return;
    }
    total += info.size;
    stat(file3, (error, info) => {
      if (error) {
```

kodefox

Readability Issues

When code readability is this bad, we're more likely to let errors sneak in.

kodefox

OK, Let's talk about Promises.

kodefox

The Promise Land

Thin but powerful abstraction on top of callbacks.

Solves several problems:

- **Easy chaining; sequential/parallel tasks**
- **Error handling**
- **Composable; can pass around a representation**

kodefox

```
// Promise style
readFile('config.json')
.then(...)
.catch(...);
```

kodefox

It's better, I Promise

In it's basic form, it looks no better than callback style.

.. but you actually get a lot more.

kodefox

Chaining Promises

```
sleep(1000)
  .then(() => {
    console.log('one');
    return sleep(1000);
  })
  .then(() => {
    console.log('two');
    return sleep(1000);
  })
  .then(() => {
    console.log('three');
  });
```

kodefox

Flow Control

We can easily combine sequential and parallel tasks to create advanced flows.

```
// This would be very difficult with callbacks
fetchJSON('/user-profile')
  .then((user) => {
    return fetchJSON(`/users/${user.id}/friends`);
  })
  .then((friendIDs) => {
    let promises = friendIDs.map((id) => {
      return fetchJSON(`/users/${id};`);
    });
    return Promise.all(promises);
  })
  .then((friends) => console.log(friends));
```

Error Handling

Attach a single catch()

Exceptions will bubble up similar to how it works in synchronous code.

```
fetchJSON('/user-profile')
  .then((user) => { ... })
  .then((friendIDs) => { ... })
  .then((friends) => { ... })
  .catch((error) => {
    console.error('And error occurred.');
});
```

kodefox

But we're still putting callbacks inside .then()

Can we do better?

kodefox

```
// What we want
let promise = fetch('/users');
// TODO: Somehow wait for the promise to resolve
console.log(promise.result);
```

kodefox

But JavaScript is fundamentally single-threaded.

So we still can’t block.

kodefox

However...

There is a special thing called a "Generator Function" that can be paused.

```
function* generatorFunc() {
  let result = fetch('/users');
  // Pause execution by yielding.
  yield result;
  // Later something caused us to resume.
  console.log(`We're back!`);
}
```

kodefox

Promises + Generators =

Awesome!

kodefox

async/await

is basically a thin layer of syntax

over Promises and Generators

kodefox

```
async function getUsers() {
  // Here's the magic
  let result = await fetchJSON('/users');
  console.log(result);
}
```

kodefox

Total Win

We get back most of our traditional constructs:

- **for/while**
- **try/catch**
- **readable, sequential program flow**
- **powerful inter-op with promises**

```
async function readConfig() {
  try {
    let content = await readFile('config.json');
    let obj = JSON.parse(content.toString());
    console.log(obj);
  } catch (error) {
    console.error('An error occurred', error);
  }
}
```

```
async function animate(element) {
  for (let i = 0; i < 100; i++) {
    element.style.left = i + 'px';
    await sleep(16);
  }
}
```

kodefox

It's just promises

- An async function always returns a promise.
- When we await a promise, our function pauses until the promise is ready (resolved)
- We can still use all our favorite promise helpers such as Promise.all()

```
async function getUserFriends() {
  let user = await fetchJSON('/users/me');
  let friendIDs = await fetchJSON(`/friends/${user.id}`);
  let promises = friendIDs.map((id) => {
    return fetchJSON(`/users/${id}`);
  });
  let friends = await Promise.all(promises);
  console.log(friends);
}

let promise = getUserFriends();
```

kodefox

Pro Tips

- Don't forget to await!
- Be careful about doing too much sequentially when you can actually do it in parallel
- Using await in map/filter won't do what you might expect!
- Even though it looks synchronous, remember your code has been paused/resumed

```
async function getNameAndAge() {
  let name = await readLine('What is your name? ');
  console.log(`Your name: ${name}`);
  let age = await readLine('What is your age? ');
  console.log(`Your age: ${age}`);
  console.log('Thank you!');
  process.exit();
}
```

kodefox

You can use this Today!

- Chrome
- Firefox
- Node
- Use Babel for the rest.

kodefox

Thanks for Listening!

simon@kodefox.com

github.com/sstur

@sstur_